

//This is an implementation in C of a preferred embodiment  
 //of the invention for the calculation of the Fourier transform

```
#include <stdio.h>

#include <math.h> //math.h is required only for the calculation of 0,+1,-1 Fourier coeffs

#define GEN 1000 //the maximum allowed generation

#define N 50 //output bandwidth

#define COR 1 //irrelevant parameter

#define EPS 1.e-12 //tolerance, where contributions to Fourier coeffs of norm less than
//EPS are regarded as negligible

#define PI 3.141592653589793

#define NVAN 1 //terminate the recursion when NVAN consecutive
//generations have contributed negligibly

#define MING 15 //calculate at least MING generations

#define SMODE 1 //SMODE=1 includes the calculation of 0,+1,-1 Fourier coeffs
//SMODE=0 is faster and does not require math.h

struct complex{ //a complex number
    double x;
    double y;
};

struct trip{ //convenient to have a real triple
    double alp;
    double bet;
    double gam;
};

struct edge{ //an arrow-structure
    int a,b,c,d;
    double e,alp,bet,gam;
};

int count=0; //a running count of the total number of samples of input data

struct complex fourier[2*N+1]; //the complex Fourier coeffs stored in the order
//n=-N,...,-1,0,1,...,N, so fourier(n) is (N+n)th coeff
```

```

double abar,bbar,cbar;           //coeffs in trig function abar*cos theta + bbar*sin theta + cbar
                                 //which agrees with input function for theta = -PI, -PI/2, 0

void main(void){

    void generotop(struct edge *,int,int,double); //recursive routine for top of circle
    void generobot(struct edge *,int,int,double); //recursive routine for bottom of circle

    void normalout(void);           //output normalization routine
    void normalin(void);           //input normalization routine

    double fbar(int,int); //fbar(p,q) returns f(theta) - (abar*cos theta + bbar*sin theta + cbar)
                           //where theta is the argument of the complex number (p-iq)/(p+iq)

    struct edge doe{1};           //initial edge for recursive calls

    int i;

    normalin();                   //this calculates the normalization parameters abar, bbar, cbar

    doe->a=doe->d=1;               //initialize the matrix of the doe to the identity
    doe->b=doe->c=0;
    doe->alp=doe->bet=doe->gam=doe->e=0; //initialize wavelet and lagged trig coeffs

    generotop(doe,1,0,fbar(-1,1)); //begin recursion on top of the circle

    generobot(doe,1,0,0.);         //begin recursion on bottom of the circle

    normalout();                  //this divides each output Fourier coeff by an overall factor PI
                                 //which was suppressed during the recursion and determines
                                 //negative from positive Fourier coeffs using reality of input

    printf("with EPS=%e, NVAN=%d, COR=%d, and MING=%d, the count is: %d\n",
           EPS,NVAN,COR,MING,count);

    for(i=0;i<=N;i++){
        printf("c[%3d] = %e+ (%i)%e\n",i,fourier[N+i].x,fourier[N+i].y);
    }

    void generotop(struct edge *p,int g,int envy,double fc){ //recursion for top of circle

        //input pointer to a current edge p of generation g, where there
        //have so far been envy consecutive generations of negligible
        //contributions, and fc is the current normalized input sample

```

```

struct edge twofer[2];           // the descendant edges of p

struct complex temp;

void genertop(struct edge *,int,int,double); //recursive function

struct complex tgener(struct edge *,struct edge *,double);

//tgener(p,twofer,f) stuffs twofer with the descendants of p
//using the updated normalized input sample f returning the
//next two normalized samples as its real and imaginary parts

int charles(struct edge *); //charles(p) updates the Fourier coeffs with the
//contribution from edge p returning 1 if they
//are negligible, and returning 0 otherwise

void tprune(struct edge *); //this implements the processing of a terminal edge

if(envy>=NVAN && g>= MING){ //if appropriate, then terminate cascade
    tprune(p);
    return;
}

temp=tgener(p,twofer,fc); //otherwise, generate descendants of p

if(charles(twofer)*charles(twofer+1)) //if both descendants contribute negligibly,
    envy++; //then increment envy
else //otherwise reset envy to zero
    envy=0;

if(g>=GEN){ //problem: non-convergence after GEN generations
    printf("fall through with gen=%d for a,b,c,d=%d,%d,%d and e=%c\n",
        g,p->a,p->b,p->c,p->d,p->e);
    return;
}

else{ //otherwise, continue recursion with descendants

    genertop(twofer,g+1,envy,temp.x);
    genertop(twofer+1,g+1,envy,temp.y);
}

}

void generbot(struct edge *p,int g,int envy,double fc){ //recursion for bottom of circle

```

```

    struct edge twofer[2];                                //this is entirely analogous to genertop

    struct complex temp;

    void generbot(struct edge *,int,int,double);
    struct complex bgener(struct edge *,struct edge *,double);
    int charles(struct edge *);
    void bprune(struct edge *);

    if(envy>=NVAN && g >= MING){
        bprune(p);
        return;
    }

    temp=bgener(p,twofer,fc);

    if(charles(twofer)*charles(twofer+1))
        envy++;
    else
        envy=0;

    if(g>=GEN){
        printf("fall through with gen=%d for a,b,c,d=%d,%d,%d,%d and e=%e\n",
            g,p->a,p->b,p->c,p->d,p->e);
        return;
    }

    else{
        generbot(twofer,g+1,envy,temp.x);
        generbot(twofer+1,g+1,envy,temp.y);
    }
}

int charles(struct edge * p){
    //charles(p) updates the Fourier coeffs using p
    //and returns a control parameter flag=1 if
    //contribution is negligible and flag=0 otherwise

    int n,flag;

    struct complex makecp(int,int); //input p,q returns the complex number (p-iq)/(p+iq)

    struct complex mult(struct complex,struct complex); //complex multiplication

    int a,b,c,d; //local variables for matrix values
    double e;     //local variable for wavelet coeff

    struct complex pzl,pzx,pzf,pzb; //l,r,f,b labels left,right,front,back vertices
    //of Farey tessellation near the edge p, and
    struct complex plm,prn,pfm,pbm; //in a loop below on index n, pzx=(pxm)^n

```

```

                                //for x=1,r,f,b
struct complex tmp,mul,cn;

int sixupc(int,int,int,int,double);                                //this updates the Fourier coeffs 0,+1,-1
                                                                    //provided SMODE=1

a=p->a;                                                                //stuff local values
b=p->b;
c=p->c;
d=p->d;
e=p->e;

pzt.x=b-d;                                                            //set-up for loop over Fourier coeffs
pzt.y=c-a;
prm=makecpX(b-d,a-c);
pzt=mult(pzt,pzt);

pzt.x=b+d;
pzt.y=-c-a;
plm=makecpX(b+d,a+c);
pzt=mult(pzt,plm);

pzt.x=d;
pzt.y=-c;
plm=makecpX(d,c);
pzt=mult(pzt,pzt);

pzt.x=b;
pzt.y=-a;
plm=makecpX(b,a);
pzt=mult(pzt,plm);

flag=1;                                                                //initialize flag
tmp.x=0;

if(SMODE)
    flag=sixupc(a,b,c,d,e);    //update 0,+1,-1 modes and re-set flag to 0 if any
                                //of these contributions is non-negligible

for(n=2;n<=N;n++){                                                    //loop over positive Fourier coeffs n>1

    tmp.y=-(n-n*n*n);

    pzt=mult(pzt,prm);    //recursively calculate powers
    pzt=mult(pzt,plm);

```

```

    pzf=mult(j...pfn);
    pzb=mult(pzb,pbm);

    cn.x=-pzt.x+2*pzf.x+2*pzb.x-pzl.x;
    cn.y=-pzt.y+2*pzf.y+2*pzb.y-pzl.y;

    mul=mult(cn,tmp);

    fourier[n+N].x+=mul.x;    //update nth Fourier coeff
    fourier[n+N].y+=mul.y;

    if(flag==1 && mul.x*mul.x+mul.y*mul.y>=EPS) //if contribution to nth Fourier
        flag=0;                               //coeff is non-negligible,
                                                //then set flag to zero
    }

    return(flag);
}

int sixupc(int a,int b,int c,int d,double e){
    //this updates the 0,+1,-1 Fourier coeffs
    //using the input matrix entries a,b,c,d
    //and wavelet coeff e

    double tp,tm,tr,tl; //p,m,r,l labels the terminal,initial,right,left vertices of Farey
                        //tessellation near edge, and tx denotes corresponding angle
    double dtemp;

    double tb;          //tb=1 on the top and tb=-1 on the bottom of the circle

    int fi;             //this is the local version of the flag in charles and is the
                        //value returned to charles by sixupc

    struct complex makecpx(int,int), temp;

    fi=1;               //set the flag to 1

    tb=1.;              //default to the top of the circle
    if(b+c<0)           //if on the bottom of the circle
        tb=-1.;         //then re-set tb=-1.

    //calculate the various angles

    temp=makecpx(d,-c);
    tp=acos(tb*temp.x);

    temp=makecpx(b,-a);

```

```

tm=acos(tb*temp.x);

temp=makecpx(b+d,-(a+c));
tl=acos(tb*temp.x);

temp=makecpx(b-d,c-a);
tr=acos(tb*temp.x);

dtemp=e*(
    2.*tp*(c*c+d*d)+2.*tm*(a*a+b*b)           //contribution to 0th Fourier coeff
    -tl*((a+c)*(a+c)+(b+d)*(b+d))
    -tr*((a-c)*(a-c)+(b-d)*(b-d))
);

if(dtemp*dtemp>=EPS) //if this contribution is non-negligible,
    fl=0;           //then re-set the flag

fourier[N].x+=dtemp; //up-date the 0th Fourier coeff

temp.x=c*(
    tp*(c*c-d*d)+tm*(a*a-b*b)           //contribution to real part of 1st
    +tr*((b-d)*(b-d)-(a-c)*(a-c))/2.    //Fourier coeff
    +tl*((b+d)*(b+d)-(a+c)*(a+c))/2.
);

fourier[N+1].x+=temp.x;                 //update real part of 1st Fourier coeff

temp.y=c*(
    2.*(tp*c*d+tm*a*b)                 //contribution to imag part of 1st
    -tr*(b-d)*(a-c)                     //Fourier coeff
    -tl*(b+d)*(a+c)
);

fourier[N+1].y+=temp.y;                 //update imag part of 1st Fourier coeff

if(temp.x*temp.x+temp.y*temp.y>=EPS) //if this contribution is non-negligible,
    fl=0;                               //then re-set the flag to zero

return(fl);
}

struct complex mult(struct complex u,struct complex v){ //complex multiplication
    struct complex temp;

```

```

        temp.x=u.x*v.x-u.y.y;
        temp.y=u.x*v.y+v.x*u.y;

        return(temp);
    }

    struct complex makecpx(int p,int q){          //input p,q returns the complex number (p+iq)/(p+iq)

        struct complex temp;
        double den;

        den=p*p+q*q;

        temp.x=(p*p-q*q)/den;
        temp.y=-2*p*q/den;

        return(temp);
    }

    void normalout(void){                          //output normalization

        double fz=0,fi=0,fo=0;
        double alp,bet,gam;
        int n;

        for(n=0;n<=N;n++){
            fourier[N+n].x=fourier[N+n].x/PI; //include factor PI suppressed before
            fourier[N+n].y=fourier[N+n].y/PI;
            fourier[N-n].x=fourier[N+n].x;    //calculate negative from positive
            fourier[N-n].y=-fourier[N+n].y;    //Fourier coeffs
        }

    }

    void normalin(void){                           //input normalization

        double fz=0,fi=0,fo=0;                  //values of input function for angles -PI, -PI/2, 0

        int i;

        double f(int,int);                      //f(p,q) returns the un-normalized input sample
                                                //at the angle corresponding to the point on the
                                                //circle given by the complex number (p-iq)/(p+iq)

        for(i=0;i<=2*N;i++){
            fourier[i].y=fourier[i].x=0;
        }
    }

```



```

    fz=f(0,1);
    fi=f(1,0);
    fo=f(1,1);

    abar=(fi-fz)/2;    //abar*cos theta + bbar*sin theta + cbar takes the same values
    cbar=(fi+fz)/2;    //as f for the angles theta=-PI, -PI/2, 0
    bbar=cbar-fo;

    fourier[N].x=cbar*PI;    //stuff the 0th Fourier coeff scaled by PI to
    fourier[N].y=0.;        //recover it after re-scaling by 1/PI in normalout

    fourier[N+1].x=abar*PI/2;    //stuff the 1st Fourier coeff scaled by PI to
    fourier[N+1].y=-bbar*PI/2;  //recover it after re-scaling by 1/PI in normalout
}

double fbar(int p, int q){
    struct complex temp;    //input p,q to fbar produces the normalized value
    struct complex makecpx(int,int);    //of the input data at the point (p-iq)/(p+iq)
    double f(int,int);
    temp=makecpx(p,q);
    return(f(p,q) - (abar*temp.x + bbar*temp.y+cbar));
}

double f(int p,int q){
    double fsinm(int,int,int);    //f is the input data, where f(p,q) is the
    return(fsinm(p,q,6));    //value taken at (p-iq)/(p+iq)
}

double fsinm(int p,int q, int m){
    int n;    //for example, the input function is
    struct complex zeta,meta;    //here taken to be sin(6*theta)
    struct complex makecpx(int,int),mult(struct complex,struct complex);

    zeta=makecpx(p,q);
    meta.x=1.;
    meta.y=0;

    for(n=1;n<=m;n++){
        meta=mult(zeta,meta);
    }

    return(meta.y);
}

struct complex tgener(struct edge *pc,struct edge *pn,double fc){
    //tgener(p, twofer,fc) stuffs twofer with the descendants
    //of p for the top of the circle, where fc is the current

```

```

//normalized input sample, and tgener returns a complex number.
//the real and imag parts being the normalized input values
//required for the least-squares regularization

double fbar(int,int);           //fbar(p,q) returns the normalized input values

int a,b,c,d,bpd,apc;           //local variables for matrix entries, bpd=b+d, apc=a+c

double e,alp,bet,gam;          //local variables for wavelet and lagged trig coeffs

double ax1,ay1,ax2,ay2;        //index 1,2 refers to first,second (counter-clockwise)
double bx1,by1,bx2,by2;        //descendants, prefix a,b,c refers to alp,bet,gam
double cx1,cy1,cx2,cy2;        //update procedures below define indices x,y

double sig1,tau1,sig2,tau2,chi; //quantities used in the least-squares regularization

double eu,et;                  //wavelet coeffs of first and second descendant edges

double fn,fnp;                 //normalized input samples

double v1c,v1u,v1t,v2c,v2u,v2t;
//coeffs in the inhomogeneous linear expression vic+viu*eu+v1t*et, i=1,2,
//of ongoing approximation to normalized input at the respective sample points
//(((2b+d)+i(2a+c))/((2b+d)-i(2a+c)), ((b+2d)+i(a+2c))/((b+2d)-i(a+2c))

struct complex temp;
struct edge *pnp;

pnp=pn;                         //initialize pointers
pnp++;

a=pc->a;                         //initialize matrix entries
b=pc->b;
c=pc->c;
d=pc->d;
apc=a+c;
bpd=b+d;

pn->a=a;                         //stuff matrix of first descendant
pn->b=b;
pn->c=apc;
pn->d=bpd;

pnp->a=apc;                      //stuff matrix of second descendant
pnp->b=bpd;
pnp->c=c;
pnp->d=d;

```

```

alp=pc->alp;           //initialize lagged trig coeffs
bet=pc->bet;
gam=pc->gam;

e=pc->e;           //initialize wavelet coeff

temp.x=fn=fbar(-(b+bpd),a+apc); //store new normalized input samples as the
temp.y=fn=fnp=fbar(-(d+bpd),c+apc); //real and imag parts of temp to return

count+=2;           //update count of total sample points

if(a>c){             //prepare for stuffing alp,bet,gam in case a>c

    ax1=alp+(d*d-c*c+2*(a*c-b*d)+a*a-b*b)*2*e;
    ay1=2*(b*b-a*a);

    bx1=bet+(c*d-a*d-b*c-a*b)*4*e;
    by1=4*a*b;

    cx1=gam+(2*(a*c+b*d)-c*c-d*d+a*a+b*b)*2*e;
    cy1=-2*(a*a+b*b);

    ax2=alp+4*(d*d-c*c)*e;
    ay2=2*(c*c-d*d);

    bx2=bet+8*e*c*d;
    by2=-4*c*d;

    cx2=gam-4*e*(c*c+d*d);
    cy2=2*(c*c+d*d);

    chi=2*e*(
        ((a+c)*(a+c)+(b+d)*(b+d))*(fc-gam)
        -alp*((b+d)*(b+d)-(a+c)*(a+c))
        -bet*2*(a+c)*(b+d)
    )/4.;
}

else{                //prepare for stuffing alp,bet,gam in case c>=a

    ax1=alp+4*e*(a*a-b*b);
    ay1=2*(b*b-a*a);

    bx1=bet-8*e*a*b;
    by1=4*a*b;

    cx1=gam+4*e*(a*a+b*b);

```

```

cy1=-2*(a-a*b*b);

ax2=alp+(a*a-b*b+2*(b*d-a*c)-c*c+d*d)*2*e;
ay2=2*(c*c-d*d);

bx2=bet+(a*d-a*b*b*c+c*d)*4*e;
by2=-4*c*d;

cx2=gam+(a*a+b*b-2*(a*c+b*d)-c*c-d*d)*2*e;
cy2=2*(c*c+d*d);

chi=-2*e*(
((a+c)*(a+c)+(b+d)*(b+d))*(fc-gam)
-alp*((b+d)*(b+d)-(a+c)*(a+c))
-bet*2*(a+c)*(b+d)
)/4.;
}
//chi=eu-ct

```

//in any case, calculate the coeffs vic,viu,vit, for i=1,2

```

v1c=-fn+cx1+
(((b+bpd)*(b+bpd)-(a+apc)*(a+apc))*ax1
+2*(b+bpd)*(a+apc)*bx1)/((b+bpd)*(b+bpd)+(a+apc)*(a+apc));

v1t=cyl+
(((b+bpd)*(b+bpd)-(a+apc)*(a+apc))*ay1
+2*by1*(b+bpd)*(a+apc))/((b+bpd)*(b+bpd)+(a+apc)*(a+apc));

v1u=8./((b+bpd)*(b+bpd)+(a+apc)*(a+apc));

v2c=-fnp+cx2+
(((d+bpd)*(d+bpd)-(c+apc)*(c+apc))*ax2
+2*(d+bpd)*(c+apc)*bx2)/((d+bpd)*(d+bpd)+(c+apc)*(c+apc));

v2u=cy2+
(((d+bpd)*(d+bpd)-(c+apc)*(c+apc))*ay2
+2*(d+bpd)*(c+apc)*by2)/((d+bpd)*(d+bpd)+(c+apc)*(c+apc));

v2t=-8./((d+bpd)*(d+bpd)+(c+apc)*(c+apc));

sig1=v1c+chi*v1u;
tau1=v1t+v1u;

sig2=v2c+chi*v2u;
tau2=v2t+v2u;

et=-(sig1*tau1+sig2*tau2)/(tau1*tau1+tau2*tau2); //calculate et and eu
eu=chi+et;

```

```

pn->e=eu;
pnp->e=et;

pn->alp=ax1+ay1*et;
pn->bet=bx1+by1*et;
pn->gam=cx1+cy1*et;

pnp->alp=ax2+ay2*eu;
pnp->bet=bx2+by2*eu;
pnp->gam=cx2+cy2*eu;

return (temp);
}

```

```

struct complex bgener(struct edge *pc, struct edge *pn, double fc){
    //bgener(p,twofer,fc) stuffs twofer with the descendants
    //of p for the bottom of the circle, where fc is the current
    //normalized input sample and returns a complex number,
    //the real and imag parts being the normalized input values
    //required for the least-squares regularization

    //this is entirely analogous to tgener, and only the differences
    //will be noted here

```

```

double fbar(int,int);

```

```

int a,b,c,d,bpd,apc;
double e,alp,bet,gam;

```

```

double ax1,ay1,ax2,ay2;
double bx1,by1,bx2,by2;
double cx1,cy1,cx2,cy2;

```

```

double sig1,tau1,sig2,tau2,chi;
double v1c,v1u,v1t,v2c,v2u,v2t;

```

```

double eu,et,fn,fnp;

```

```

struct complex temp;

```

```

struct edge *pnp;

```

```

pnp=pn;
pnp++;

```

```

a=pc->d;           //differs from tgener in that d,-c,-b,a replaces a,b,c,d
b=-(pc->c);
c=-(pc->b);
d=pc->a;

apc=a+c;
bpd=b+d;

pn->a=bpd;          //differs from tgener in that d,-c,-b,a replaces a,b,c,d
pn->b=-apc;
pn->c=-b;
pn->d=a;

pnp->a=d;           //differs from tgener in that d,-c,-b,a replaces a,b,c,d
pnp->b=-c;
pnp->c=-bpd;
pnp->d=apc;

alp=pc->alp;
bet=pc->bet;
gam=pc->gam;

c=pc->e;

temp.x=fn=fbar(a+apc,b+bpd); //differs from tgener in that
temp.y=fnp=fbar(c+apc,d+bpd); //(-q,p) replaces (p,q)

count+=2;

if(a>c){

    ax1=alp+(d*d-c*c+2*(a*c-b*d)+a*a-b*b)*2*e;
    ay1=2*(b*b-a*a);

    bx1=bct+(c*d-a*d-b*c-a*b)*4*e;
    by1=4*a*b;

    cx1=gam+(2*(a*c+b*d)-c*c-d*d+a*a+b*b)*2*e;
    cy1=-2*(a*a+b*b);

    ax2=alp+4*(d*d-c*c)*e;
    ay2=2*(c*c-d*d);

    bx2=bet+8*c*c*d;
    by2=-4*c*d;

    cx2=gam-4*e*(c*c+d*d);
    cy2=2*(c*c+d*d);

```

```

        chi=2*e+(
        ((a+c)*(a+c)+(b+d)*(b+d))*(fc-gam)
        -alp*((b+d)*(b+d)-(a+c)*(a+c))
        -bet*2*(a+c)*(b+d)
        )/4.;
    }

else{

    ax1=alp+4*e*(a*a-b*b);
    ay1=2*(b*b-a*a);

    bx1=bet-8*e*a*b;
    by1=4*a*b;

    cx1=gam+4*e*(a*a+b*b);
    cy1=-2*(a*a+b*b);

    ax2=alp+(a*a-b*b+2*(b*d-a*c)-c*c+d*d)*2*e;
    ay2=2*(c*c-d*d);

    bx2=bet+(a*d-a*b+b*c+c*d)*4*e;
    by2=-4*c*d;

    cx2=gam+(a*a+b*b-2*(a*c+b*d)-c*c-d*d)*2*e;
    cy2=2*(c*c+d*d);

    chi=-2*e+(
    ((a+c)*(a+c)+(b+d)*(b+d))*(fc-gam)
    -alp*((b+d)*(b+d)-(a+c)*(a+c))
    -bet*2*(a+c)*(b+d)
    )/4.;
    }

    vic=-fm+cx1+
    (((b+bpd)*(b+bpd)-(a+apc)*(a+apc))*ax1
    +2*(b+bpd)*(a+apc)*bx1)/((b+bpd)*(b+bpd)+(a+apc)*(a+apc));

    vlt=cy1+
    (((b+bpd)*(b+bpd)-(a+apc)*(a+apc))*ay1
    +2*by1*(b+bpd)*(a+apc))/((b+bpd)*(b+bpd)+(a+apc)*(a+apc));

    vlu=3./((b+bpd)*(b+bpd)+(a+apc)*(a+apc));

```

```

v2c=-fnp+cx2+
(((d+bpd)*(d+bpd)-(c+apc)*(c+apc))*ax2
+2*(d+bpd)*(c+apc)*bx2)/((d+bpd)*(d+bpd)+(c+apc)*(c+apc));

```

```

v2u=cy2+
(((d+bpd)*(d+bpd)-(c+apc)*(c+apc))*ay2
+2*(d+bpd)*(c+apc)*by2)/((d+bpd)*(d+bpd)+(c+apc)*(c+apc));

```

```

v2t=-8./((d+bpd)*(d+bpd)+(c+apc)*(c+apc));

```

```

sig1=v1c+chi*v1u;
tau1=v1t+v1u;

```

```

sig2=v2c+chi*v2u;
tau2=v2t+v2u;

```

```

et=-(sig1*tau1+sig2*tau2)/(tau1*tau1+tau2*tau2);
eu=chi+et;

```

```

pn->e=eu;
pnp->e=et;

```

```

pn->alp=ax1+ay1*et;
pn->bet=bx1+by1*et;
pn->gam=cx1+cy1*et;

```

```

pnp->alp=ax2+ay2*eu;

```

```

pnp->bet=bx2+by2*eu;
pnp->gam=cx2+cy2*eu;

```

```

return (temp);
}

```

```

void tprune(struct edge *p){
    //this routine implements the processing of terminal
    //edge p in the cascade for the top of the circle

    double alp, bet, gam,e; //local copies of lagged trig coeffs and wavelet coeff

    double alp1,alp2,bet1,bet2,gam1,gam2; //trig coeffs of the trigonometric function sigma at
    //the first and second (counter-clockwise) samples

    struct trip mob(struct complex,struct complex, struct complex,
    double,double,double);
    //mob(z1,z2,z3,f1,f2,f3) returns the triple with entries alp,bet,gam
    //where alp*cos theta + bet*sin theta + gam takes the respective values
    // f1,f2,f3 at the points z1,z2,z3 on the circle given as complex numbers

```



```

double fbar(int,int);           //returns normalized input values

void fixup(struct trip,int,int,int,int); //adjusts the output Fourier coeffs using sigma-tau

struct complex makecpX(int,int); //input p,q returns (p-iq)/(p+iq)

void sixup(struct trip,int,int,int,int,int); //adjusts the 0,+1,-1 Fourier coeffs provided
//SMODE=1

int a,b,c,d;                   //local variables for the matrix entries

struct trip trip1,trip2;       //first and second trig triples

a=p->a;                         //stuff local variables
b=p->b;
c=p->c;
d=p->d;
e=p->e;
alp=p->alp;
bet=p->bet;
gam=p->gam;

if (a>c) { //final update of trig fields in case a>c
    alp1=alp+e*2*(a*(a+c)+d*(d-b)-b*(b+d)-c*(c-a));
    bet1=bet+e*4*(c*(d-b)-a*(d+b));
    gam1=gam+e*2*(a*(a+c)+c*(a-c)+b*(b+d)-d*(d-b));

    alp2=alp+e*4*(d-c)*(d+c);
    bet2=bet+e*8*c*d;
    gam2=gam-e*4*(c*c+d*d);
}

else { //final update of trig fields in case c>=a
    alp1=alp+e*4*(a-b)*(a+b);
    bet1=bet-e*8*a*b;
    gam1=gam+e*4*(a*a+b*b);

    alp2=alp+e*2*(a*(a-c)+b*(d-b)-c*(c+a)+d*(b+d));
    bet2=bet+e*4*(a*(d-b)+c*(b+d));
    gam2=gam+e*2*(a*(a-c)-c*(a-c)-b*(d-b)-d*(b+d));
}

trip1=mob( //calculate sigma for first descendant
makecpX(-(3*b+d),3*a+c),
makecpX(-(2*b+d),2*a+c),
makecpX(-(3*b+2*d),3*a+2*c),
fbar(-(3*b+d),3*a+c),
fbar(-(2*b+d),2*a+c),

```

```

        fbar(-(3*b+2*d),3*a+2*c)
    );

    trip2=mob(                                //calculate sigma for second descendant
        makecpx(-(2*b+3*d),2*a+3*c),
        makecpx(-(b+2*d),a+2*c),
        makecpx(-(b+3*d),a+3*c),
        fbar(-(2*b+3*d),2*a+3*c),
        fbar(-(b+2*d),a+2*c),
        fbar(-(b+3*d),a+3*c)
    );

    count+=6;                                //update count of total sample points

    trip1.alp=(trip1.alp-alp1)/COR;           //difference of first trig functions
    trip1.bet=(trip1.bet-bet1)/COR;
    trip1.gam=(trip1.gam-gam1)/COR;

    trip2.alp=(trip2.alp-alp2)/COR;           //difference of second trig functions
    trip2.bet=(trip2.bet-bet2)/COR;
    trip2.gam=(trip2.gam-gam2)/COR;

    fixup(trip1,b+d,a+c,b,a,1); //adjust Fourier coeffs with first trig function difference

    fixup(trip2,d,c,b+d,a+c,1); //adjust Fourier coeffs with second trig function difference

    if(SMODE){                                //if SMODE=1, then adjust 0,+1,-1 Fourier coeffs as well
        sixupp(trip1,b+d,a+c,b,a,1);
        sixupp(trip2,d,c,b+d,a+c,1);
    }
}

void bprune(struct edge *p){                  //this routine implements the processing of terminal
                                              //edge p in the cascade for the bottom of the circle

                                              //this is entirely analogous to tgener, and only the
                                              //differences will be noted here

    double alp, bet, gam,e;
    double alp1,alp2,bet1,bet2,gam1,gam2;
    struct trip mob(struct complex,struct complex, struct complex,
                    double,double,double);

    double fbar(int,int);
    void fixup(struct trip, int,int,int,int,int);
    struct complex makecpx(int,int);
    void sixupp(struct trip,int,int,int,int,int);
    int a,b,c,d;
    struct trip trip1,trip2;

```

```

a=p->d;           //differs from tprune in that d,-c,-b,a replaces a,b,c,d
b=(p->c);
c=(p->b);
d=p->a;
e=p->e;
alp=p->alp;
bet=p->bet;
gam=p->gam;

if (a>c){
    alp1=alp+e*2*(a*(a+c)+d*(d-b)-b*(b+d)-c*(c-a));
    bet1=bet+e*4*(c*(d-b)-a*(d+b));
    gam1=gam+e*2*(a*(a+c)+c*(a-c)+b*(b+d)-d*(d-b));

    alp2=alp+e*4*(d-c)*(d+c);
    bet2=bet+c*8*c*d;
    gam2=gam-e*4*(c*c+d*d);
}

else{
    alp1=alp+e*4*(a-b)*(a+b);
    bet1=bet-c*8*a*b;
    gam1=gam+e*4*(a*a+b*b);

    alp2=alp+c*2*(a*(a-c)+b*(d-b)-c*(c+a)+d*(b+d));
    bet2=bet+e*4*(a*(d-b)+c*(b+d));
    gam2=gam+e*2*(a*(a-c)-c*(a+c)-b*(d-b)-d*(b+d));
}

trip1=mob(           //arguments of mob differ from those in tprune
    makecpx(3*a+c,3*b+d),
    makecpx(2*a+c,2*b+d),
    makecpx(3*a+2*c,3*b+2*d),
    fbar(3*a+c,3*b+d),
    fbar(2*a+c,2*b+d),
    fbar(3*a+2*c,3*b+2*d)
);

trip2=mob(           //arguments of mob differ from those in tprune
    makecpx(2*a+3*c,2*b+3*d),
    makecpx(a+2*c,b+2*d),
    makecpx(a+3*c,b+3*d),
    fbar(2*a+3*c,2*b+3*d),
    fbar(a+2*c,b+2*d),
    fbar(a+3*c,b+3*d)
);

count+=6;

```

```

trip1.alp=(trip1.alp+alp1)/COR; //differs from tprune in that alp,bet
trip1.bet=(trip1.bet+bet1)/COR; //is replaced by -alp,-bet
trip1.gam=(trip1.gam-gam1)/COR;

trip2.alp=(trip2.alp+alp2)/COR; //differs from tprune in that alp,bet
trip2.bet=(trip2.bet+bet2)/COR; //is replaced by -alp,-bet
trip2.gam=(trip2.gam-gam2)/COR;

fixup(trip1,b+d,a+c,b,a,-1); //last argument of fixup differs from tprune
fixup(trip2,d,c,b+d,a+c,-1);Q //last argument of fixup differs from tprune

if(SMODE){ //if SMODE=1, adjust the 0,+1,-1 Fourier coeffs

    sixupp(trip1,b+d,a+c,b,a,-1); //last argument of sixupp differs from tprune
    sixupp(trip2,d,c,b+d,a+c,-1); //last argument of sixupp differs from tprune
}

}

struct trip mob(struct complex z1,struct complex z2,struct complex z3,
               double f1,double f2,double f3){
    //mob(z1,z2,z3,f1,f2,f3) returns the triple with entries alp,bet,gam
    //where alp*cos theta + bet*sin theta + gam takes the respective values
    // f1,f2,f3 at the points z1,z2,z3 on the circle given as complex numbers

    double det,aa,bb,cc,dd;
    struct trip temp;

    aa=(z1.x-z3.x);
    bb=(z1.y-z3.y);
    cc=(z2.x-z3.x);
    dd=(z2.y-z3.y);

    det=aa*dd-bb*cc; //general principles guarantee that det is non-zero

    aa=aa/det;
    bb=bb/det;
    cc=cc/det;
    dd=dd/det;

    temp.alp=dd*(f1-f3)-bb*(f2-f3);
    temp.bet=aa*(f2-f3)-cc*(f1-f3);
    temp.gam=f3-z3.x*temp.alp-z3.y*temp.bet;

    return(temp);
}

```

```

void fixup(struct trip delt,int dm,int cm,int dp,int cp,int tb){

    //fixup delt,dm,cm,dp,cp,tb) adjusts the Fourier coeffs by adding the Fourier
    //coeffs of  $\text{delt} = T^{\wedge} C \cdot T$  with support truncated to be the interval with endpoints
    //(dm+i*cm)/(dm-i*cm), (dp+i*dp)/(dp-i*cp)

    //let tm, tp be the corresponding angles

    // tb=+1 on the top, and tb=-1 on the bottom of the circle

    int n;

    double cpp,cmp,cpm,cmm;
    //in loop below on n, cpx=[cos(1+n)tx]/(n+1), cmx=[cos(1-n)tx]/(1-n), for x=p,m

    double spm,smm, spp,smg;
    //in loop below on n, spx=[sin(1+n)tx]/(1+n), smx=[sin(1-n)tx]/(1-n), for x=p,m

    double crm,crp,srm,srp;
    //in loop below on n, crm=[cos n*tx]/n, srm=[sin n*tx]/n, for x=p,m

    double alp,bet,gam; //local variables for entries of delt

    double ctp,ctm,cnp,cnm; //in loop below on n, ctx=cos tx, cnx=cos n*tx, for x=p,m
    double stn,snm,stp,snp; //in loop below on n, stx=sin tx, snx=sin n*tx, for x=p,m

    double cbufp,sbufp,cbufm,sbufm;
    struct complex makecpx(int,int),temp;

    alp=delt.alp; //stuff local variables
    bet=delt.bet;
    gam=delt.gam;

    temp=makecpx(dm,-cm);
    ctm=tb*temp.x;
    stm=tb*temp.y;

    temp=makecpx(dp,-cp);
    ctp=tb*temp.x;
    stp=tb*temp.y;

    cnp=ctp*ctm-stp*stm;
    snp=2*stp*ctp;

    cnm=ctm*ctm-stm*stm;
    snm=2*stm*ctm;

```

```

for(n=2;n<=N;n++){
    //loop over Fourier coeffs
    cbufp=ctp*cnp-stp*snp; //update using formulas for cos
    sbufp=stp*cnp+ctp*snp; //for sin of a sum of angles
    cbufm=ctm*cnm-stm*snm;
    sbufm=stm*cnm+ctm*snm;

    cpp=cbufp/(1+n); //update for p
    cmp=(cbufp+2*stp*snp)/(1-n);
    spp=sbufp/(1+n);
    smp=(sbufp-2*ctp*snp)/(1-n);

    cpm=cbufm/(1+n); //update for m
    cmm=(cbufm+2*stm*snm)/(1-n);
    spm=sbufm/(1+n);
    smm=(sbufm-2*ctm*snm)/(1-n);

    crp=2.*cnp/n;
    crm=2.*cnm/n;
    srp=2.*snp/n;
    srm=2.*snm/n;

    cnp=cbufp; //update for next iteration
    snp=sbufp;
    cnm=cbufm;
    snm=sbufm;

    fourier[N+n].x+=( //adjust real part of nth Fourier coeff
        +alp*(smp+spp-smm-spm)
        +bet*(-cpp-cmp+cpm+cmm)
        +gam*(srp-srm)
    )/4.;

    fourier[N+n].y+=( //adjust imag part of nth Fourier coeff
        +alp*(cpp-cmp-cpm+cmm)
        +bet*(spp-smp-spm+smm)
        +gam*(crp-crm)
    )/4.;
}

return;
}

```

```

void sixup(struct trip triple,int dm,int cm,int dp,int cp,int tb){

```

//sixup(delt,dm,cm,dp,cp,tb) adjusts the 0,+1,-1 Fourier coeffs by adding the 0,+1,-1 Fourier  
 //coeffs of delt=T°C-T with support truncated to be the interval with endpoints

```

//((dm+i*cm)/(dm-i*cm), (dp+i*dp)/(dp-i*cp)

//let tm, tp be the corresponding angles

// tb=+1 on the top, and tb=-1 on the bottom of the circle

double alp,bet,gam; //local variables for entries of delt
double com,sim,cop,sip,tm,tp; //cox=cos tx, six=sin tx, for x=p,m

struct complex temp;
struct complex makecpX(int,int);

temp=makecpX(dm,-cm); //calculate com,sim,tm
com=tb*temp.x;
sim=tb*temp.y;
tm=acos(com);

temp=makecpX(dp,-cp); //calculate cop,sip,tp
cop=tb*temp.x;
sip=tb*temp.y;
tp=acos(cop);

alp=tb*triple.alp; //adjust alp,bet for the bottom of the circle
bet=tb*triple.bet;
gam=triple.gam;

fourier[N].x+=( //update 0th Fourier coeff
    gam*(tp-tm)+alp*(sip-sim)-bet*(cop-com)
)/2.;

fourier[N+1].x+=( //update real part of 1st Fourier coeff
    alp*(tp-tm)/2.
    +gam*(sip-sim)
    -bet*(cop*cop-sip*sip-com*com+sim*sim)/4.
    +alp*(cop*sip-com*sim)/2.
)/2.;

fourier[N+1].y+=( //update imag part of 1st Fourier coeff
    -bet*(tp-tm)/2.
    +gam*(cop-com)
    +alp*(cop*cop-sip*sip-com*com+sim*sim)/4.
    +bet*(cop*sip-com*sim)/2.
)/2.;
}

```

//Here are the subroutines replacing tgener and bgener above in an  
 //implementation which takes advantage of renormalization

```
struct complex tgener(struct edge *pc,struct edge *pn,double fc)
{
```

```
    double fbar(int,int);
    int a,b,c,d,bpd,apc;
    double e,alp,bet,gam;
    double ax1,ay1,ax2,ay2;
    double bx1,by1,bx2,by2;
    double cx1,cy1,cx2,cy2;
    double sig1,tau1,sig2,tau2,chi;
    double v1c,v1u,v1t,v2c,v2u,v2t;
    double deriv(int,int,int,int,int,int);
    double dc,dn,dnp;
    double eu,et,fa,fnp;
    struct complex temp;
    struct edge *pnp;
    extern int count;
```

```
    pnp=pn;
    pnp++;
```

```
    a=pc->a;
    b=pc->b;
    c=pc->c;
    d=pc->d;
```

```
    apc=a+c;
    bpd=b+d;
```

```
    pn->a=a;
    pn->b=b;
    pn->c=apc;
    pn->d=bpd;
```

```
    pnp->a=apc;
    pnp->b=bpd;
    pnp->c=c;
    pnp->d=d;
```



```

alp=pc->alp;
bet=pc->bet;
gam=pc->gam;

```

```

e=pc->e;

```

```

temp.x=fn=fbar(-(b+bpd),a+apc);
temp.y=fnp=fbar(-(d+bpd),c+apc);
temp.n=0;

```

```

count+=2;

```

```

dc=deriv(a,b,c,d,-1,1);
dn=deriv(a,b,c,d,-1,2);
dnp=deriv(a,b,c,d,-2,1);

```

```

fc=fc/dc;
fn=fn/dn;
fnp=fnp/dnp;

```

```

if(a>c){
    ax1=alp+4.*e;
    bx1=bet-4.*e;
    cx1=gam;
    ax2=alp+4.*e;
    bx2=bet;
    cx2=gam-4.*e;
    chi=2.*((fc-gam-bet)/4.+e);
}

```

```

else{

```

```

    ax1=alp+4.*e;
    bx1=bet;
    cx1=gam+4.*e;
    ax2=alp+4.*e;
    bx2=bet+4.*e;
    cx2=gam;
    chi=2.*((fc-gam-bet)/4.-e);
}

```

```

v1c=-fn+cx1+(4.*bx1-3.*ax1)/5.;
v1t=4./5.;
v1u=8./5.;

```

```

v2c=-fnp+cx2+(4.*u^2+3.*ax2)/5.;
v2u=4./5.;
v2t=-8./5.;

sig1=v1c+chi*v1u;
tau1=v1t+v1u;

sig2=v2c+chi*v2u;
tau2=v2t+v2u;

et=(-(sig1*tau1*(1)+sig2*tau2*(1)))/(tau1*tau1*(1)
+tau2*tau2*(1));

eu=chi+et;

pn->e=eu;
pnp->e=et;

alp=ax1-2.*et;
bet=bx1;
gam=cx1-2.*et;

pn->alp=(2.*bet+alp+gam)/2.;
pn->bet=(bet-alp+gam);
pn->gam=(2.*bet-alp+3.*gam)/2.;

alp=ax2-2.*eu;
bet=bx2;
gam=cx2+2.*eu;

pnp->alp=(-2.*bet+alp-gam)/2.;
pnp->bet=alp+bet+gam;
pnp->gam=(2.*bet+alp+3.*gam)/2.;

return (temp);

```

```

}

```

```

double deriv(int a,int b,int c,int d,int p,int q){

```

```

    double x,y,nep;

    x=p*p-q*q;
    y=-2*p*q;
    x=x/(p*p+q*q);
    y=y/(p*p+q*q);
    nep=-(a*a+b*b+c*c+d*d)+x*(a*a+b*b-c*c-d*d)-y*(2*a*c+2*b*d);
    return(-2./nep);

```

```

    }

    struct complex bgener(struct edge *pc, struct edge *pn, double fc)
    {
        double fbar(int,int);
        int a,b,c,d,bpd,apc;
        double e,alp,bet,gam;
        double ax1,ay1,ax2,ay2;
        double bx1,by1,bx2,by2;
        double cx1,cy1,cx2,cy2;
        double sig1,tau1,sig2,tau2,chi;
        double v1c,v1u,v1t,v2c,v2u,v2t;
        double eu,et,fn,fnp;
        extern int count;
        double deriv(int,int,int,int,int,int);
        double dc,dn,dnp;
        struct complex temp;
        struct edge *pnp;

        pnp=pn;
        pnp++;

        a=pc->d;
        b=-(pc->c);
        c=-(pc->b);
        d=pc->a;

        apc=a+c;
        bpd=b+d;

        pn->a=bpd;
        pn->b=-apc;
        pn->c=-b;
        pn->d=a;

        pnp->a=d;
        pnp->b=-c;
        pnp->c=-bpd;
        pnp->d=apc;

        alp=pc->alp;
        bet=pc->bet;
        gam=pc->gam;

        e=pc->e;

        temp.x=fn=fbar(a+apc,b+bpd);
        temp.y=fnp=fbar(c+apc,d+bpd);
        temp.n=0;
    }

```

```

count+=2;

dc=deriv(d,-c,-b,a,1,1);
dn=deriv(d,-c,-b,a,2,1);
dnp=deriv(d,-c,-b,a,1,2);

fc=fc/dc;
fn=fn/dn;
fnp=fnp/dnp;

if(a>c){
    ax1=alp+4.*e;
    bx1=bet-4.*e;
    cx1=gam;

    ax2=alp+4.*e;
    bx2=bet;
    cx2=gam-4.*e;

    chi=2.*((fc-gam-bet)/4.+e);
}

else{
    ax1=alp+4.*e;
    bx1=bet;
    cx1=gam+4.*e;

    ax2=alp+4.*e;
    bx2=bet+4.*e;
    cx2=gam;

    chi=2.*((fc-gam-bet)/4.-e);
}

v1c=-fn+cx1+(4.*bx1-3.*ax1)/5.;
v1t=-4./5.;
v1u=8./5.;

v2c=-fnp+cx2+(4.*bx2+3.*ax2)/5.;
v2u=4./5.;
v2t=-8./5.;

sig1=v1c+chi*v1u;
tau1=v1t+v1u;

sig2=v2c+chi*v2u;
tau2=v2t+v2u;

```

```
et=(sig1*tau1*(1)...g2*tau2*(1))/(tau1*tau1*(1)
+tau2*tau2*(1));
```

```
eu=chi+et;
```

```
pn->e=eu;
```

```
pn->p=et;
```

```
alp=ax1-2.*et;
```

```
bet=bx1;
```

```
gam=cx1-2.*et;
```

```
pn->alp=(2.*bet+alp+gam)/2.;
```

```
pn->bet=(bet+alp+gam);
```

```
pn->gam=(2.*bet+alp+3.*gam)/2.;
```

```
alp=ax2-2.*eu;
```

```
bet=bx2;
```

```
gam=cx2+2.*eu;
```

```
pn->alp=(-2.*bet+alp+gam)/2.;
```

```
pn->bet=alp+bet+gam;
```

```
pn->gam=(2.*bet+alp+3.*gam)/2.;
```

```
return (temp);
```

```
}
```

//This is an implementation in C of a preferred embodiment  
 //of the invention for the calculation of the inverse Fourier transform

```
#include <stdio.h>

#define SCAT 50          //SCAT=[exp sinh^(-1)(1/SCALE)]^2
#define N 50            //input bandwidth
#define MINMODE 0       //minimum frequency of input
#define PI 3.141592653589793
#define CHUNK 5000      //chunk of memory allocated as required

struct complex{          //a complex number
    double x;
    double y;
};

struct edge{             //a complex arrow-structure
    int a,b,c,d;
    struct complex e,alp,bet,gam;
};

struct pqxy{             //the basic data type of the output, where
    int p,q;             //p,q corresponds to the complex number (p-iq)/(p+iq)
    double x,y;          //at which the output function takes the complex value x+iy
};

struct complex pfourier[2*N+1];

struct complex *fourier=pfourier+N; //the input Fourier coeffs
//indexed -N,...,-1,0,1,...,N

struct complex czer,cone,comon;      //the modified 0,+1,-1 Fourier coeffs

struct pqxy graf[CHUNK];              //output spatial values in
//clockwise-order around the circle
//starting from the complex number -1

int outcount=0;                       //running tally of the total
//number of output values

void main(void){

    void genertop(struct edge *,int); //recursive routine for top of circle
    void generbot(struct edge *,int); //recursive routine for bottom of circle

    void normalout(void);              //output normalization routine
    void normalin(void);              //input normalization routine

    struct complex cogi(struct edge *); //returns complex wavelet coeff of argument
```

```

struct complex ei,emu,emt,eu,et; //wavelet coeffs on  $1, U^{\{-1\}}, T^{\{-1\}}, U, T$ 

struct complex emitt,emtu,emut,emuui; //wavelet coeffs on  $T^{\{-2\}},$ 
// $T^{\{-1\}}U^{\{-1\}}, U^{\{-1\}}T^{\{-1\}}, U^{\{-2\}}$ 

struct edge start[1]; //initial edge for each of
//the various recursive calls

int i;

//begin input data

for(i=0;i<=2*N;i++){
    pfourier[i].x=0.;
    pfourier[i].y=0.;
}

fourier[5].x=.5;
fourier[-5].x=.5;

//end input data

normalin(); //this stuffs czer,cone,cmon with the
//modified 0,+1,-1 Fourier coefficients

graf[outcount].p=0; //initial normalized output value
graf[outcount].q=1;
graf[outcount].x=0.;
graf[outcount].y=0.;
outcount++;

//calculate the wavelet coeffs for the various required edges of small generation

start->a=start->d=1;
start->c=0;
start->b=-1;
emt=cogi(start);

start->a=start->d=1;
start->c=0;
start->b=1;
et=cogi(start);

start->a=start->d=1;
start->c=-1;
start->b=0;
emu=cogi(start);

start->a=start->d=1;
start->c=1;
start->b=0;
eu=cogi(start);

start->a=start->d=1;
start->b=start->c=0;
ei=cogi(start);

```

```

start->a=start->d=1;
start->b=0;
start->c=-2;
emuu=cogi(start);

```

```

start->a=start->d=1;
start->b=-2;
start->c=0;
emtt=cogi(start);

```

```

start->a=1;
start->b=start->c=-1;
start->d=2;
emut=cogi(start);

```

```

start->a=2;
start->d=1;
start->b=start->c=-1;
emtu=cogi(start);

```

//serially perform the recursions for the initial edges in counter-clockwise order

```

start->a=start->d=1; //initialize
start->c=1;
start->b=0;
start->e=eu;
start->alp.x=2.*ei.x-2.*et.x;
start->alp.y=2.*ei.y-2.*et.y;
start->bet.x=2.*(emt.x-emu.x)-2.*ei.x;
start->bet.y=2.*(emt.y-emu.y)-2.*ei.y;
start->gam.x=2.*ei.x-2.*et.x;
start->gam.y=2.*ei.y-2.*et.y;

```

genertop(start,1); //recursion for U

```

start->a=start->d=1; //initialize
start->c=0;
start->b=1;
start->e=et;
start->alp.x=2.*ei.x-2.*eu.x;
start->alp.y=2.*ei.y-2.*eu.y;
start->bet.x=2.*(emt.x-emu.x)+2.*ei.x;
start->bet.y=2.*(emt.y-emu.y)+2.*ei.y;
start->gam.x=2.*ei.x+2.*eu.x;
start->gam.y=2.*ei.y+2.*eu.y;

```

genertop(start,1); //recursion for T

```

start->a=start->d=1; //initialize
start->c=0;
start->b=-2;
start->e=emtt;
start->alp.x=(-2.*ei.x+2.*emu.x-4.*emt.x+2.*emut.x);
start->alp.y=(-2.*ei.y+2.*emu.y-4.*emt.y+2.*emut.y);

```



```

start->bet.x=(-2.*ei.x-2.*emu.x+2.*emt.x);
start->bet.y=(-2.*ei.y-2.*emu.y+2.*emt.y);
start->gam.x=2.*ei.x-2.*emu.x+4.*emt.x-2.*emut.x;
start->gam.y=2.*ei.y-2.*emu.y+4.*emt.y-2.*emut.y;

generbot(start,1);                                //recursion for T^{-2}

start->a=1;                                         //initialize
start->d=2;
start->b=start->c=-1;
start->e=emut;
start->alp.x=(-2.*ei.x+2.*emu.x+2.*emt.x);
start->alp.y=(-2.*ei.y+2.*emu.y+2.*emt.y);
start->bet.x=(-2.*ei.x-2.*emu.x-6.*emt.x+4.*emut.x);
start->bet.y=(-2.*ei.y-2.*emu.y-6.*emt.y+4.*emut.y);
start->gam.x=2.*ei.x-2.*emu.x-6.*emt.x+4.*emut.x;
start->gam.y=2.*ei.y-2.*emu.y-6.*emt.y+4.*emut.y;

generbot(start,1);                                //recursion for U^{\{-1\}}T^{\{-1\}}

start->a=2;                                         //initialize
start->d=1;
start->b=start->c=-1;
start->e=emtu;
start->alp.x=(-2.*ei.x+2.*emu.x+2.*emt.x);
start->alp.y=(-2.*ei.y+2.*emu.y+2.*emt.y);
start->bet.x=(-2.*ei.x+6.*emu.x+2.*emt.x-4.*emuu.x);
start->bet.y=(-2.*ei.y+6.*emu.y+2.*emt.y-4.*emuu.y);
start->gam.x=2.*ei.x+6.*emu.x+2.*emt.x-4.*emuu.x;
start->gam.y=2.*ei.y+6.*emu.y+2.*emt.y-4.*emuu.y;

generbot(start,1);                                //recursion for T^{\{-1\}}U^{\{-1\}}

start->a=start->d=1;                                //initialize
start->b=0;
start->c=-2;
start->e=emuu;
start->alp.x=(-2.*ei.x-4.*emu.x+2.*emt.x+2.*emtu.x);
start->alp.y=(-2.*ei.y-4.*emu.y+2.*emt.y+2.*emtu.y);
start->bet.x=(-2.*ei.x-2.*emu.x+2.*emt.x);
start->bet.y=(-2.*ei.y-2.*emu.y+2.*emt.y);
start->gam.x=-2.*ei.x-4.*emu.x+2.*emt.x+2.*emtu.x;
start->gam.y=-2.*ei.y-4.*emu.y+2.*emt.y+2.*emtu.y;

generbot(start,1);                                //recursion for U^{\{-2\}}

//recursions complete, and graf is stuffed with normalized output data in counter-clockwise order

normalout();                                       //un-normalize by altering
                                                    //each output value using the
                                                    //modified 0,+1,-1 Fourier coeffs

printf("cmon=%e +(i) %e\n czer=%e +(i) %e\n cone=%e +(i) %e\n\n",
       cmon.x,cmon.y,czer.x,czer.y,cone.x,cone.y);

for (i=0;i<outcount;i++)

```

```

printf("(%.4d) %e+i)%e\n",
      graf[i].p.graf[i].q.graf[i].x.graf[i].y);
}

```

```

void genertop(struct edge *p,int g){ //recursion for top of the circle
    struct edge twofer[2],*ped; //recursion generates twofer[2] from p, where
                                //the entries of twofer are the descendants of p

```

```

    struct complex temp;
    struct complex makecp(x(int,int); //input p,q returns (p-iq)/(p+iq)
    double ct,st; //cos,sin for various angles
    void genertop(struct edge *,int); //recursive function

```

```

    void tgener(struct edge *,struct edge *); //tgener(p,twofer) stuffs twofer[2]
                                                //with the descendants of p

```

```

    int subtend(struct edge *),pp,qq; //returns 1 if both argument edges
                                      //subtend angles less than SCALE,
                                      //otherwise returns 0

```

```

    tgener(p,twofer); //tgener(p,twofer) stuffs twofer
                      //with the two descendants of p

```

```

    if(subtend(twofer)){ //if both descendants subtend
                          //angles less than SCALE, then
                          //write two output values and
                          //terminate the recursion

```

```

        ped=twofer;
        pp=-(ped->d); //stuff output values for the first
        qq=ped->c; // (counter-clockwise) sample point
        graf[outcount].p=pp;
        graf[outcount].q=qq;
        temp=makecp(pp,qq);
        ct=temp.x;
        st=temp.y;
        graf[outcount].x=ped->alp.x*ct+ped->bet.x*st
            +ped->gam.x+(ped->e.x)*4./(pp*pp+qq*qq);
        graf[outcount].y=ped->alp.y*ct+ped->bet.y*st
            +ped->gam.y+(ped->e.y)*4./(pp*pp+qq*qq);
        outcount++; //update number of output values

```

```

        ped++;
        pp=-(ped->d); //stuff output values for the second
        qq=ped->c; // (counter-clockwise) sample point
        graf[outcount].p=pp;
        graf[outcount].q=qq;
        temp=makecp(pp,qq);
        ct=temp.x;
        st=temp.y;
        graf[outcount].x=ped->alp.x*ct+ped->bet.x*st
            +ped->gam.x;
        graf[outcount].y=ped->alp.y*ct+ped->bet.y*st
            +ped->gam.y;
        outcount++; //update number of output values

        return; //terminate the recursion
    }

```

```

    genertop(twofer,g+1); //in the contrary case that one of

```

```

    genertop(twofer+1,g--);

    }

    void generbot(struct edge *p,int g){

        struct edge twofer[2]*ped;

        struct complex temp, makecp(int,int);
        double ct,st;
        void generbot(struct edge *,int);
        void bgener(struct edge *,struct edge *);
        int subtend(struct edge *),pp,qq;

        bgener(p,twofer);

        if(subtend(twofer)){

            ped=twofer;
            pp=ped->b;
            qq=ped->a;
            graf[outcount].p=pp;
            graf[outcount].q=qq;
            temp=makecp(pp,qq);
            ct=temp.x;
            st=temp.y;
            graf[outcount].x=-ped->alp.x*ct-ped->bet.x*st
                +ped->gam.x+(ped->e.x)*4./(pp*pp+qq*qq); // differs from
            graf[outcount].y=-ped->alp.y*ct-ped->bet.y*st
                +ped->gam.y+(ped->e.y)*4./(pp*pp+qq*qq); // differs from
            outcount++; // b,a replaces d,c

            ped++;
            pp=ped->b;
            qq=ped->a;
            graf[outcount].p=pp;
            graf[outcount].q=qq;
            temp=makecp(pp,qq);
            ct=temp.x;
            st=temp.y;
            graf[outcount].x=-ped->alp.x*ct-ped->bet.x*st
                +ped->gam.x; // differs from genertop
            graf[outcount].y=-ped->alp.y*ct-ped->bet.y*st
                +ped->gam.y; // in that alp and bet are
            outcount++; // multiplied by -1

            return;
        }

        generbot(twofer,g+1);
        generbot(twofer+1,g+1);

    }

    struct complex mult(struct complex u,struct complex v){
        struct complex temp;

```

//multiplication  
//of complex numbers

```

temp.x=u.x*v.x-u.y.v.y;
temp.y=u.x*v.y+v.x*u.y;

return(temp);
}

struct complex makecpz(int p,int q){
    struct complex temp;
    double den;

    den=p*p+q*q;

    temp.x=(p*p-q*q)/den;
    temp.y=-2*p*q/den;

    return(temp);
}

int subtend(struct edge *p){
    int nn;

    nn=(p->a)*(p->c)+(p->b)*(p->d);
    if(nn*nn<SCAT)
        return(0);

    p++;

    nn=(p->a)*(p->c)+(p->b)*(p->d);
    if(nn*nn<SCAT)
        return(0);

    return(1);
}

void normalin(void){
    struct complex cz[4],co[4],cm[4];

    struct complex temp;

    int n;

    cz[0].x=-1.;
    cz[0].y=0.;
    cz[1].x=0.;
    cz[1].y=0.;
    cz[2].x=-1.;
    cz[2].y=0.;
    cz[3].x=0.;
    cz[3].y=0.;

    co[0].x=0.;
    co[0].y=0.;
    co[1].x=-1.;
    co[1].y=0.;

```

*//input p,q returns*  
*//the complex number*  
 *//(p-iq)/(p+iq)*

*//subtend returns 1 if each*  
*//of the two edges in the*  
*//array p subtends an angle*  
*//approximately less than*  
*//SCALE and returns 0 otherwise*

*//this routine calculates the modified*  
*//0,+1,-1 Fourier coeffs czero,cone,cmom*  
*//from the input Fourier coeffs*

*//initialize array for czero calculation*

*//initialize array for cone calculation*

```

co[2].x=0.;
co[2].y=1.;
co[3].x=0.;
co[3].y=0.;

cm[0].x=0.;           //initialize array for cmon calculation
cm[0].y=0.;
cm[1].x=0.;
cm[1].y=0.;
cm[2].x=0.;
cm[2].y=-1.;
cm[3].x=-1.;
cm[3].y=0.;

czer=fourier[0];
cone=fourier[1];
cmon=fourier[-1];

for(n=2;n<=N;n++){

    temp=mult(fourier[n],cz[n%4]);
    czer.x+=-temp.x;           //contribution to czer from the
    czer.y+=-temp.y;           //positive Fourier coeffs

    temp=mult(fourier[-n],cz[(4*N-n)%4]);

    czer.x+=-temp.x;           //contribution to czer from the
    czer.y+=-temp.y;           //negative Fourier coeffs

    temp=mult(fourier[n],co[n%4]);

    cone.x+=-temp.x;           //contribution to cone from the
    cone.y+=-temp.y;           //positive Fourier coeffs

    temp=mult(fourier[-n],co[(4*N-n)%4]);

    cone.x+=-temp.x;           //contribution to cone from the
    cone.y+=-temp.y;           //negative Fourier coeffs

    temp=mult(fourier[n],cm[n%4]);

    cmon.x+=-temp.x;           //contribution to cmon from the
    cmon.y+=-temp.y;           //positive Fourier coeffs

    temp=mult(fourier[-n],cm[(4*N-n)%4]);

    cmon.x+=-temp.x;           //contribution to cmon from the
    cmon.y+=-temp.y;           //negative Fourier coeffs
}

}

void normalout(void){
    int n,m;
    struct complex temp,temp1,makecpx(int,int);
    double ct,st;

    //this routine alters the
    //output values in the
    //array graf by adding
    //czero+cone*z+cmon*z^(-1),

```

```

//where  $z=(p-iq)/(p-iq)$ 
for (n=0;n<outcount;n++){

    temp=makecpx(graft[n].p,graft[n].q);
    temp1=mult(temp,cone);

    temp.y=-temp.y;
    temp=mult(temp,cm0n);

    graft[n].x+=czer.x+temp1.x+temp.x;
    graft[n].y+=czer.y+temp1.y+temp.y;

}

}

struct complex cogi(struct edge *p){    //cogi(p) calculates the wavelet coeff for the
                                        //arrow underlying edge p from the Fourier coeffs

    int a,b,c,d,n;
    struct complex makecpx(int,int), mult(struct complex,struct complex);
    struct complex temp2,temp1,temp,crun,cb1,cm1,cb2,cm2;

    a=p->a;
    b=p->b;
    c=p->c;
    d=p->d;
    crun.x=0.;
    crun.y=0.;

    cm1=makecpx(b,-a);
    cm2=makecpx(d,-c);

    cb1=cm1;
    cb2=cm2;

    for(n=2;n<=N;n++){
        cb1=mult(cb1,cm1);    //updated nth power of cm1
        cb2=mult(cb2,cm2);    //updated nth power of cm2

        if(n<MINMODE)//skip contribution to wavelet coeffs
            continue;        //from low-frequency Fourier coeffs

        temp.x=n;
        temp.y=b*d+a*c;
        temp1=mult(cb1,temp);
        temp.y=-temp.y;
        temp2=mult(cb2,temp);
        temp.x=temp1.x+temp2.x;
        temp.y=temp1.y+temp2.y;
        temp1=mult(fourier[n],temp);
        crun.x+=temp1.x;    //contribution to wavelet coeff from
        crun.y+=temp1.y;    //nth Fourier coeff for n positive

        temp.x=-temp.x;
        temp2=mult(fourier[-n],temp);
        crun.x+=temp2.x;    //contribution to wavelet coeff from
        crun.y+=temp2.y;    //nth Fourier coeff for n negative

    }
}

```

```

temp=crun;
crun.x=temp.y/4.; //overall multiple of 1/4
crun.y=temp.x/4.;

return(crun);
}

void tgener(struct edge *pc,struct edge *pn){ //tgener(p,twofer) stuffs twofer
                                              //with descendants of p for top
                                              //of circle

    int a,b,c,d,bpd,apc; //matrix entries a,b,c,d,
                          //bpd=b+d and apc=a+c

    struct complex e,alp,bet,gam; //wavelet coefficient e, trig coefficients alp,bet,gam

    struct complex ax1,ax2; //index 1,2 refers to first,second
    struct complex bx1,bx2; //(counter-clockwise) descendants
    struct complex cx1,cx2; //prefix a,b,c refers to alp,bet,gam
    int ay1,ay2,by1,by2,cy1,cy2; //update procedure below defines x,y

    struct complex eu,et; //wavelet coeffs of first, second
                          //descendant edges

    struct complex temp;

    struct complex cogi(struct edge *),mult(struct complex,struct complex);
    struct complex makecpx(int,int);

    double ct,st;

    struct edge *pnp;

    pnp=pn; //Initialize pointers
    pnp++;

    a=pc->a; //initialize matrix entries
    b=pc->b;
    c=pc->c;
    d=pc->d;

    apc=a+c;
    bpd=b+d;

    pn->a=a; //stuff matrix of first descendant
    pn->b=b;
    pn->c=apc;
    pn->d=bpd;

    pnp->a=apc; //stuff matrix of descendant
    pnp->b=bpd;
    pnp->c=c;
    pnp->d=d;

```

```

alp=pc->alp;           //initialize lagged trig coeffs
bet=pc->bet;
gam=pc->gam;

e=pc->e;               //initialize wavelet coeff

if(a>c){
    //prepare for stuffing alp,bet,gam in case a>c

    ax1.x=alp.x+(d*d-c*c+2*(a*c-b*d)+a*a-b*b)*2*e.x;
    ax1.y=alp.y+(d*d-c*c+2*(a*c-b*d)+a*a-b*b)*2*e.y;

    ay1=2*(b*b-a*a);

    bx1.x=bet.x+(c*d-a*d-b*c-a*b)*4*e.x;
    bx1.y=bet.y+(c*d-a*d-b*c-a*b)*4*e.y;

    by1=4*a*b;

    cx1.x=gam.x+(2*(a*c+b*d)-c*c-d*d+a*a+b*b)*2*e.x;
    cx1.y=gam.y+(2*(a*c+b*d)-c*c-d*d+a*a+b*b)*2*e.y;

    cy1=-2*(a*a+b*b);

    ax2.x=alp.x+4*(d*d-c*c)*e.x;
    ax2.y=alp.y+4*(d*d-c*c)*e.y;

    ay2=2*(c*c-d*d);

    bx2.x=bet.x+8*e.x*c*d;
    bx2.y=bet.y+8*e.y*c*d;

    by2=-4*c*d;

    cx2.x=gam.x-4*e.x*(c*c+d*d);
    cx2.y=gam.y-4*e.y*(c*c+d*d);

    cy2=2*(c*c+d*d);
}

else{
    //prepare for stuffing alp,bet,gam in case c>=a

    ax1.x=alp.x+4*e.x*(a*a-b*b);
    ax1.y=alp.y+4*e.y*(a*a-b*b);

    ay1=2*(b*b-a*a);

    bx1.x=bet.x-8*e.x*a*b;
    bx1.y=bet.y-8*e.y*a*b;

    by1=4*a*b;

    cx1.x=gam.x+4*e.x*(a*a+b*b);
    cx1.y=gam.y+4*e.y*(a*a+b*b);

    cy1=-2*(a*a+b*b);

    ax2.x=alp.x+(a*a-b*b+2*(b*d-a*c)-c*c+d*d)*2*e.x;

```



```
ax2.y=alp.*(a*a-b*b+2*(b*d-a*c)-c*c+d*d)*2*e.y;
```

```
ay2=2*(c*c-d*d);
```

```
bx2.x=bet.x+(a*d-a*b+b*c+c*d)*4*e.x;
```

```
bx2.y=bet.y+(a*d-a*b+b*c+c*d)*4*e.y;
```

```
by2=-4*c*d;
```

```
cx2.x=gam.x+(a*a+b*b-2*(a*c+b*d)-c*c-d*d)*2*e.x;
```

```
cx2.y=gam.y+(a*a+b*b-2*(a*c+b*d)-c*c-d*d)*2*e.y;
```

```
cy2=2*(c*c+d*d);
```

```
}
```

```
pn->e=cogit(pn); //calculate wavelet coeffs
```

```
pnp->e=et=cogit(pnp); //of the descendants
```

```
pn->alp.x=ax1.x+ay1*et.x; //update first alp
```

```
pn->alp.y=ax1.y+ay1*et.y;
```

```
pn->bet.x=bx1.x+by1*et.x; //update first bet
```

```
pn->bet.y=bx1.y+by1*et.y;
```

```
pn->gam.x=cx1.x+cy1*et.x; //update first gam
```

```
pn->gam.y=cx1.y+cy1*et.y;
```

```
pnp->alp.x=ax2.x+ay2*eu.x; //update second alp
```

```
pnp->alp.y=ax2.y+ay2*eu.y;
```

```
pnp->bet.x=bx2.x+by2*eu.x; //update second bet
```

```
pnp->bet.y=bx2.y+by2*eu.y;
```

```
pnp->gam.x=cx2.x+cy2*eu.x; //update second gam
```

```
pnp->gam.y=cx2.y+cy2*eu.y;
```

```
}
```

```
void bgener(struct edge *pc,struct edge *pn){
```

```
//bgener(p,twofer) stuffs twofer
```

```
//with descendants of p for
```

```
//bottom of circle,
```

```
//it is entirely analogous to
```

```
//tgener, and only the
```

```
//differences will be noted here
```

```
struct complex ax1,ax2;
```

```
struct complex bx1,bx2;
```

```
struct complex cx1,cx2;
```

```
int ay1,ay2,by1,by2,cy1,cy2;
```

```
struct complex eu,et;
```

```

struct complex ten;

struct complex cogi(struct edge *),mult(struct complex,struct complex);
struct complex makecpx(int,int);

struct edge *pnp;

double ct,st;

pnp=pn;
pnp++;

a=pc->d;          //differs from tgener in that d,-c,-b,a
b=-(pc->c);        //replaces a,b,c,d
c=(pc->b);
d=pc->a;

apc=a+c;
bpd=b+d;

pn->a=bpd;         //differs from tgener in that d,-c,-b,a
pn->b=apc;         //replaces a,b,c,d
pn->c=-b;
pn->d=a;

pnp->a=d;          //differs from tgener in that d,-c,-b,a
pnp->b=-c;         //replaces a,b,c,d
pnp->c=-bpd;
pnp->d=apc;

alp=pc->alp;
bet=pc->bet;
gam=pc->gam;

e=pc->c;

if(a>c){

    ax1.x=alp.x+(d*d-c*c+2*(a*c-b*d)+a*a-b*b)*2*e.x;
    ax1.y=alp.y+(d*d-c*c+2*(a*c-b*d)+a*a-b*b)*2*e.y;

    ay1=2*(b*b-a*a);

    bx1.x=bet.x+(c*d-a*d-b*c-a*b)*4*e.x;
    bx1.y=bet.y+(c*d-a*d-b*c-a*b)*4*e.y;

    by1=4*a*b;

    cx1.x=gam.x+(2*(a*c+b*d)-c*c-d*d+a*a+b*b)*2*e.x;
    cx1.y=gam.y+(2*(a*c+b*d)-c*c-d*d+a*a+b*b)*2*e.y;

    cy1=2*(a*a+b*b);

    ax2.x=alp.x+4*(d*d-c*c)*e.x;
    ax2.y=alp.y+4*(d*d-c*c)*e.y;

    ay2=2*(c*c-d*d);

    bx2.x=bet.x+8*e.x*c*d;

```

```

    bx2.y=bet.x-.8*e.y*c*d;

    by2=-4*c*d;

    cx2.x=gam.x-4*e.x*(c*c+d*d);
    cx2.y=gam.y-4*e.y*(c*c+d*d);

    cy2=2*(c*c+d*d);
}

else{
    ax1.x=alp.x+4*e.x*(a*a-b*b);
    ax1.y=alp.y+4*e.y*(a*a-b*b);

    ay1=2*(b*b-a*a);

    bx1.x=bet.x-8*e.x*a*b;
    bx1.y=bet.y-8*e.y*a*b;

    by1=4*a*b;

    cx1.x=gam.x+4*e.x*(a*a+b*b);
    cx1.y=gam.y+4*e.y*(a*a+b*b);

    cy1=-2*(a*a+b*b);

    ax2.x=alp.x+(a*a-b*b+2*(b*d-a*c)-c*c+d*d)*2*e.x;
    ax2.y=alp.y+(a*a-b*b+2*(b*d-a*c)-c*c+d*d)*2*e.y;

    ay2=2*(c*c-d*d);

    bx2.x=bet.x+(a*d-a*b+b*c+c*d)*4*e.x;
    bx2.y=bet.y+(a*d-a*b+b*c+c*d)*4*e.y;

    by2=-4*c*d;

    cx2.x=gam.x+(a*a+b*b-2*(a*c+b*d)-c*c-d*d)*2*e.x;
    cx2.y=gam.y+(a*a+b*b-2*(a*c+b*d)-c*c-d*d)*2*e.y;

    cy2=2*(c*c+d*d);
}

pn->c=eu=cog(pn);
pnp->e=et=cog(pnp);

pn->alp.x=ax1.x+ay1*et.x;
pn->alp.y=ax1.y+ay1*et.y;

pn->bet.x=bx1.x+by1*et.x;
pn->bet.y=bx1.y+by1*et.y;

pn->gam.x=cx1.x+cy1*et.x;
pn->gam.y=cx1.y+cy1*et.y;

pnp->alp.x=ax2.x+ay2*eu.x;
pnp->alp.y=ax2.y+ay2*eu.y;

pnp->bet.x=bx2.x+by2*eu.x;

```

```

pnp->bet.y=bx2.y+_+_eu.y;

pnp->gam.x=cx2.x+cy2*eu.x;
pnp->gam.y=cx2.y+cy2*eu.y;
}

```

//Here are the subroutines replacing tgener and bgener above in an  
//implementation which takes advantage of renormalization

```

void tgener(struct edge *pc,struct edge *pn)
{

```

```

    int a,b,c,d,bpd,apc;
    struct complex e,alp,bet,gam;

    extern int outcount;
    extern struct pqxy graf[CHUNK];

    struct complex ax1,ax2;
    struct complex bx1,bx2;
    struct complex cx1,cx2;
    struct complex eu,et;
    struct complex cogi(struct edge *);

```

```

    double ct,st;
    struct edge *pnp;

```

```

    pnp=pn;
    pnp++;
    a=pc->a;
    b=pc->b;
    c=pc->c;
    d=pc->d;

```

```

    apc=a+c;
    bpd=b+d;

```

```

    pn->a=a;
    pn->b=b;
    pn->c=apc;
    pn->d=bpd;

```

```

    pnp->a=apc;
    pnp->b=bpd;
    pnp->c=c;
    pnp->d=d;

```

```

    alp=pc->alp;
    bet=pc->bet;
    gam=pc->gam;

```

```

    c=pc->c;

```

```

if(a>c){
    ax1.x=alp.x+4.*e.x;
    ax1.y=alp.y+4.*e.y;
    bx1.x=bet.x-4.*e.x;
    bx1.y=bet.y-4.*e.y;
    cx1.x=gam.x;
    cx1.y=gam.y;

    ax2.x=ax1.x;
    ax2.y=ax1.y;
    bx2.x=bet.x;
    bx2.y=bet.y;
    cx2.x=gam.x-4.*e.x;
    cx2.y=gam.y-4.*e.y;
}

else{
    ax1.x=alp.x+4.*e.x;
    ax1.y=alp.y+4.*e.y;
    bx1.x=bet.x;
    bx1.y=bet.y;
    cx1.x=gam.x+4.*e.x;
    cx1.y=gam.y+4.*e.y;

    ax2.x=ax1.x;
    ax2.y=ax1.y;
    bx2.x=bet.x+4.*e.x;
    bx2.y=bet.y+4.*e.y;
    cx2.x=gam.x;
    cx2.y=gam.y;
}

pn->e=cu=cogi(pn);
pnp->e=et=cogi(pnp);

alp.x=ax1.x-2.*et.x;
bet.x=bx1.x;
gam.x=cx1.x-2.*et.x;

alp.y=ax1.y-2.*et.y;
bet.y=bx1.y;
gam.y=cx1.y-2.*et.y;

pn->alp.x=(2.*bet.x+alp.x+gam.x)/2.;
pn->bet.x=(bet.x-alp.x+gam.x);
pn->gam.x=(2.*bet.x-alp.x+3.*gam.x)/2.;

pn->alp.y=(2.*bet.y+alp.y+gam.y)/2.;
pn->bet.y=(bet.y-alp.y+gam.y);
pn->gam.y=(2.*bet.y-alp.y+3.*gam.y)/2.;

alp.x=ax2.x-2.*eu.x;
bet.x=bx2.x;
gam.x=cx2.x+2.*eu.x;

alp.y=ax2.y-2.*eu.y;

```

```

bet.y=bx2.y;
gam.y=cx2.y+2.*eu.y;

pnp->alp.x=(-2.*bet.x+alp.x-gam.x)/2.;
pnp->bet.x=alp.x+bet.x+gam.x;
pnp->gam.x=(2.*bet.x+alp.x+3.*gam.x)/2.;

pnp->alp.y=(-2.*bet.y+alp.y-gam.y)/2.;
pnp->bet.y=alp.y+bet.y+gam.y;
pnp->gam.y=(2.*bet.y+alp.y+3.*gam.y)/2.;

}

double deriv(int a,int b,int c,int d,int p,int q){

    double x,y,nep;

    x=p*p*q*q;
    y=2*p*p*q;
    x=x/(p*p+q*q);
    y=y/(p*p+q*q);
    nep=-(a*a+b*b+c*c+d*d)+x*(a*a+b*b-c*c-d*d)-y*2.*(a*c+b*d);
    return(-2./nep);

}

void bgener(struct edge *pc,struct edge *pn)
{

    int a,b,c,d,bpd,apc;
    struct complex e,alp,bet,gam;
    extern int outcount;
    extern struct pqxy graf[CHUNK];
    struct complex ax1,ax2;
    struct complex bx1,bx2;
    struct complex cx1,cx2;
    struct complex eu,et;
    struct complex cogi(struct edge *);
    double ct,st;
    struct edge *pnp;

    pnp=pn;
    pnp++;

    a=pc->d;
    b=(pc->c);
    c=(pc->b);
    d=pc->a;

    apc=a+c;
    bpd=b+d;

    pn->a=bpd;
    pn->b=apc;
    pn->c=b;
    pn->d=a;

```

```

pnp->a=d;
pnp->b=-c;
pnp->c=-bpd;
pnp->d=apc;

```

```

alp=pc->alp;
bet=pc->bet;
gam=pc->gam;

```

```

e=pc->e;

```

```

if(a>c){
    ax1.x=alp.x+4.*e.x;
    ax1.y=alp.y+4.*e.y;
    bx1.x=bet.x-4.*e.x;
    bx1.y=bet.y-4.*e.y;
    cx1.x=gam.x;
    cx1.y=gam.y;

    ax2.x=ax1.x;
    ax2.y=ax1.y;
    bx2.x=bet.x;
    bx2.y=bet.y;
    cx2.x=gam.x-4.*e.x;
    cx2.y=gam.y-4.*e.y;
}

```

```

else{
    ax1.x=alp.x+4.*e.x;
    ax1.y=alp.y+4.*e.y;
    bx1.x=bet.x;
    bx1.y=bet.y;
    cx1.x=gam.x+4.*e.x;
    cx1.y=gam.y+4.*e.y;

    ax2.x=ax1.x;
    ax2.y=ax1.y;
    bx2.x=bet.x+4.*e.x;
    bx2.y=bet.y+4.*e.y;
    cx2.x=gam.x;
    cx2.y=gam.y;
}

```

```

pn->e=eu=cogi(pn);
pnp->e=et=cogi(pnp);

```

```

pn->c=eu;
pnp->c=et;

```

```

alp.x=ax1.x-2.*et.x;
bet.x=bx1.x;
gam.x=cx1.x-2.*et.x;

```

```

alp.y=ax1.y-2.*et.y;
bet.y=bx1.y;

```

```
gam.y=cx1.y-2.*eu.x;

pn->alp.x=(2.*bet.x+alp.x+gam.x)/2.;
pn->bet.x=(bet.x-alp.x+gam.x);
pn->gam.x=(2.*bet.x-alp.x+3.*gam.x)/2.;

pn->alp.y=(2.*bet.y+alp.y+gam.y)/2.;
pn->bet.y=(bet.y-alp.y+gam.y);
pn->gam.y=(2.*bet.y-alp.y+3.*gam.y)/2.;

alp.x=ax2.x-2.*eu.x;
bet.x=bx2.x;
gam.x=cx2.x+2.*eu.x;

alp.y=ax2.y-2.*eu.y;
bet.y=bx2.y;
gam.y=cx2.y+2.*eu.y;

pnp->alp.x=(-2.*bet.x+alp.x-gam.x)/2.;
pnp->bet.x=alp.x+bet.x+gam.x;
pnp->gam.x=(2.*bet.x+alp.x+3.*gam.x)/2.;

pnp->alp.y=(-2.*bet.y+alp.y-gam.y)/2.;
pnp->bet.y=alp.y+bet.y+gam.y;
pnp->gam.y=(2.*bet.y+alp.y+3.*gam.y)/2.;
```